CS 348 Lecture 3-1

Relational Model Part 2

Semih Salihoğlu

Jan 13th, 2025

Outline

- More examples of relational algebra
- Monotone operators
- Relational calculus
- SQL (second half of lecture)

(Recap) Relational data model

- A database is a collection of relations (or tables)
- Each relation has a set of attributes (or columns)
- Each attribute has a unique name and a domain (or type)
  - The domains are required to be **atomic**

Single, indivisible piece of information

- Each relation contains a set of tuples (or rows)
  - Each tuple has a value for each attribute of the relation
  - Duplicate tuples are not allowed
    - Two tuples are duplicates if they agree on all attributes

☞Simplicity is a virtue!

(Recap) Integrity constraints

- Candidate key
  - Set of K attributes that uniquely identify a row and has only the necessary attributed (i.e., minimal)
- Primary key
- Foreign key

(Recap) RA operators

Core Operators

1. Selection: $\sigma_p R$
2. Projection: $\pi_L R$
3. Cross product: $R \times S$
4. Union: $R \cup S$
5. Difference: $R - S$
6. Renaming: $\rho_{S(A_1 \rightarrow A_1', A_2 \rightarrow A_2', \ldots)} R$

Derived Operators

1. Join: $R \bowtie_p S$
2. Natural join: $R \bowtie S$
3. Intersection: $R \cap S$

More example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- All groups (ids) that Lisa belongs to

More example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- All groups (ids) that Lisa belongs to

Writing a query bottom-up:

uid	name	age	pop
857	Lisa	8	0.7

Who's Lisa? $\sigma_{name="Lisa"}$

$User$

uid	*name*	*age*	*pop*
123	Milhouse	10	0.2
857	Lisa	8	0.7
…	…	…	…

$Member$

uid	*gid*
123	gov
857	abc
857	gov
…	…

More example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- All groups (ids) that Lisa belongs to

Writing a query bottom-up:

uid	name	age	pop	gid
857	Lisa	8	0.7	abc
857	Lisa	8	0.7	gov

$\bowtie_{\text{User.uid=Member.uid}}$

uid	name	age	pop
857	Lisa	8	0.7

Who's Lisa? $\sigma_{name="Lisa"}$

$User$

$Member$

uid	*gid*
123	gov
857	abc
857	gov
…	…

More example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- All groups (ids) that Lisa belongs to

Writing a query bottom-up:

Lisa's groups π_{gid}

gid
abc
gov

uid	name	age	pop	gid
857	Lisa	8	0.7	abc
857	Lisa	8	0.7	gov

$\bowtie_{User.uid=Member.uid}$

uid	name	age	pop
857	Lisa	8	0.7

Who's Lisa? $\sigma_{name="Lisa"}$

$User$

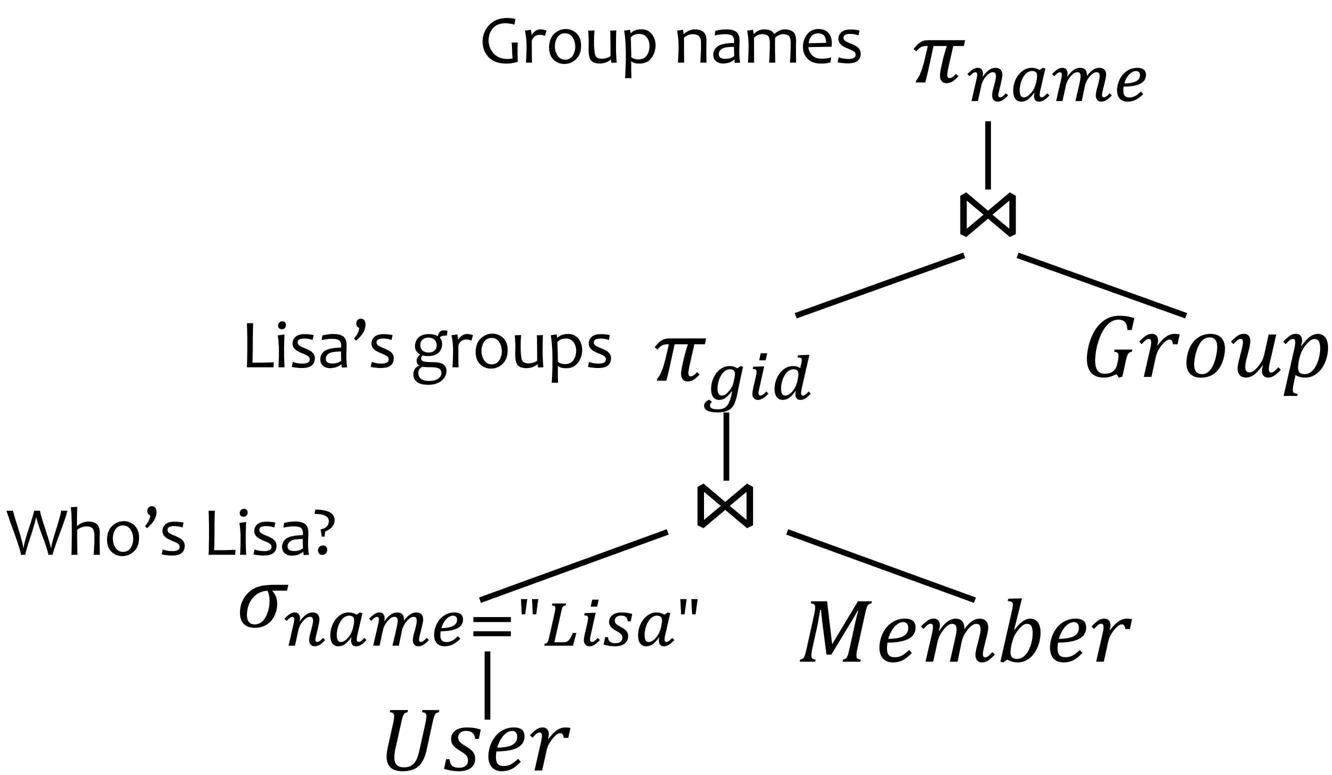
$Member$

uid	gid
123	gov
857	abc
857	gov
…	…

Take home ex.

User (uid int, *name* string, *age* int, *pop* float)
Group (gid string, *name* string)
Member (uid int, gid string)

- All groups ~~(ids)~~ that Lisa belongs to
  names?



More example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

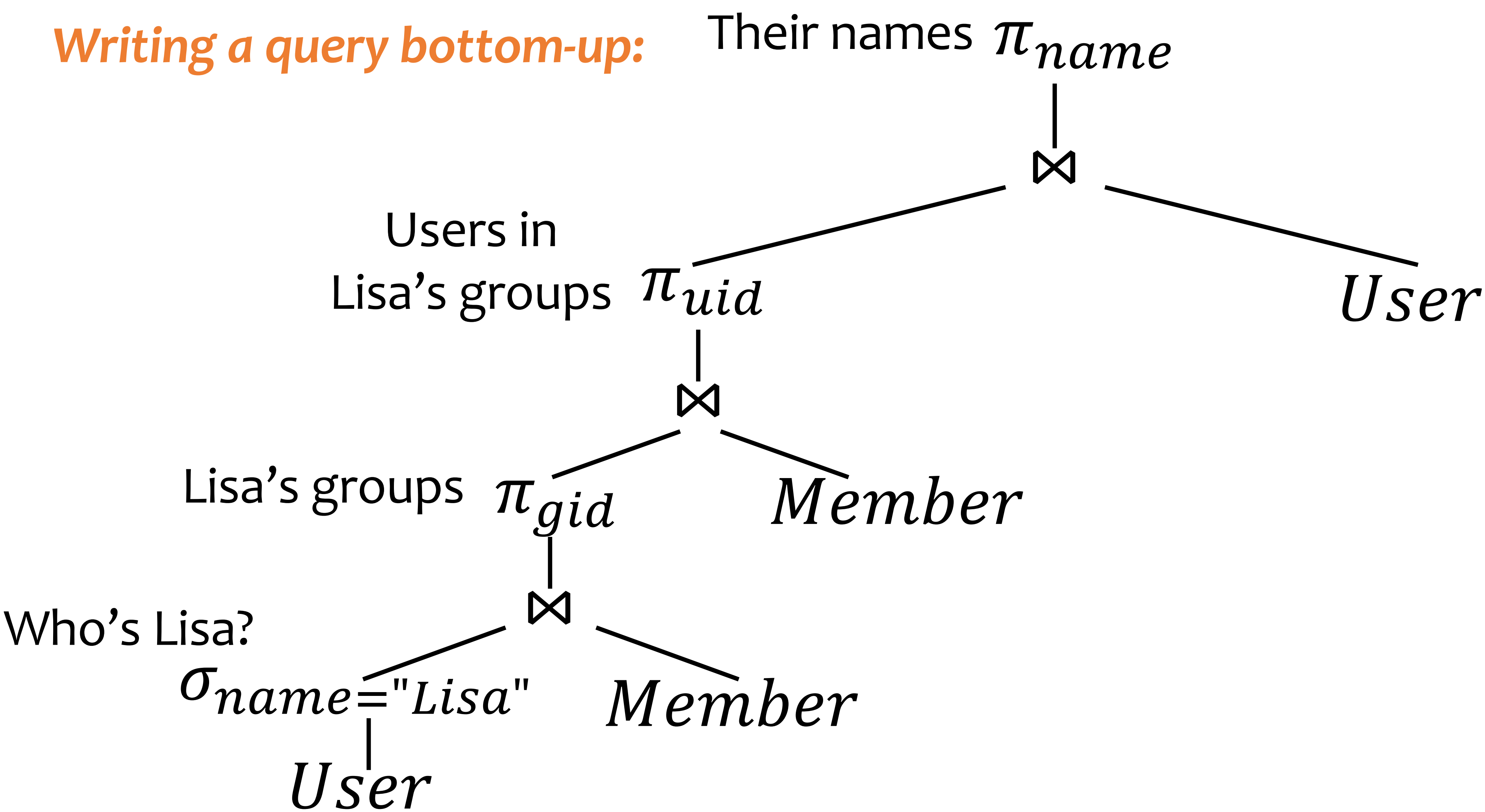
- Names of users in Lisa's groups

More example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- Names of users in Lisa's groups

Writing a query bottom-up:



More example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- IDs of groups that Lisa doesn't belong to

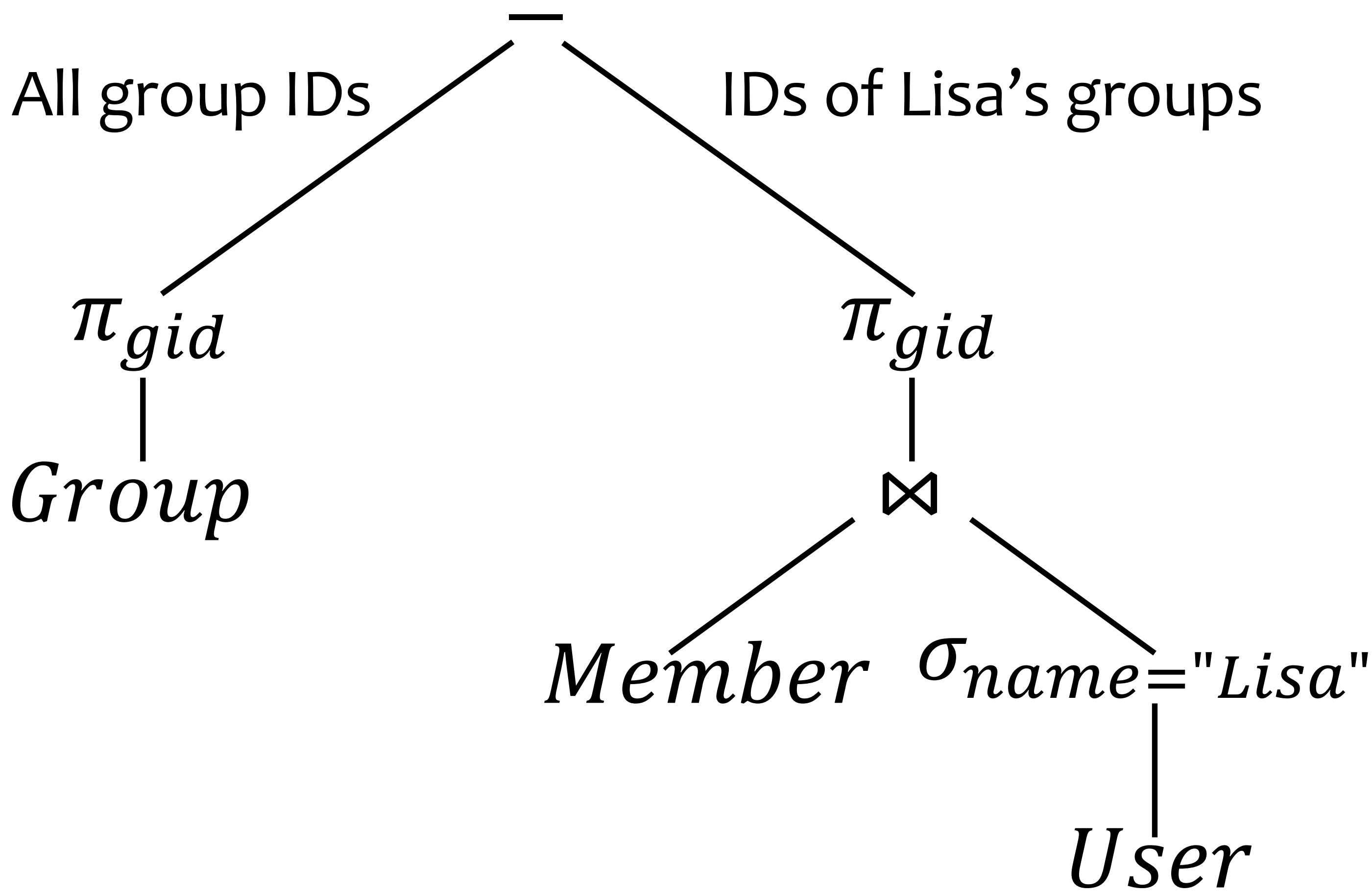
Writing a query top-down:

More example

User (uid int, *name* string, *age* int, *pop* float)
Group (gid string, *name* string)
Member (uid int, gid string)

- IDs of groups that Lisa doesn't belong to

Writing a query top-down:



A trickier example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- Who are the most popular users?

$$\sigma_{pop \geq every\ pop\ in\ User}\ User$$ **WRONG!**

- Because it cannot be evaluated over a single row

A trickier example

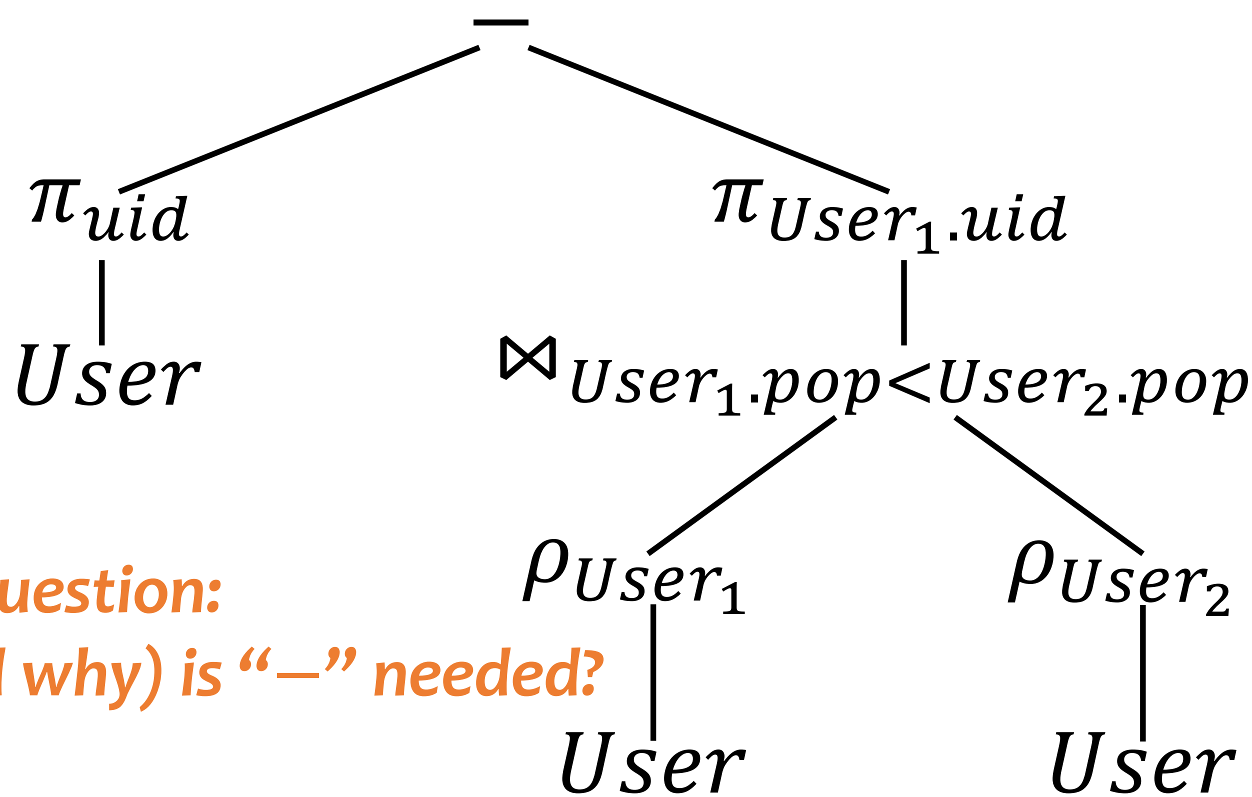
User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

- Who are the most popular users?
  - Who do NOT have the highest *pop* rating?
  - Whose *pop* is lower than somebody else's?

A trickier example

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

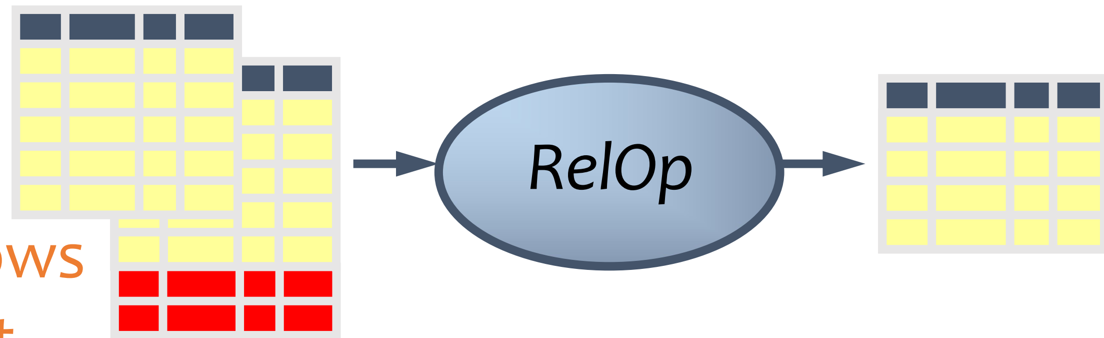
- Who are the most popular users?
  - Who do NOT have the highest *pop* rating?
  - Whose *pop* is lower than somebody else's?



A deeper question:
When (and why) is "—" needed?

Non-monotone operators

Add more rows to the input...



What happens to the output?

- If some old output rows may become invalid → the operator is non-monotone

- Example: difference operator $R - S$

uid	*gid*
123	gov
857	abc

R

$-$

uid	*gid*
123	gov
901	edf
857	**abc**

S

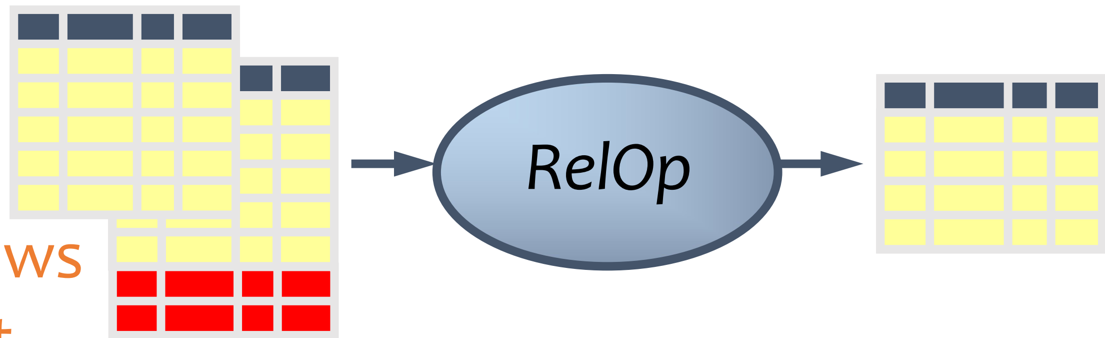
=

uid	*gid*
~~857~~	~~abc~~

This old row becomes invalid because the new row added to S

Non-monotone operators

Add more rows to the input…



What happens to the output?

- If some old output rows may become invalid (causing some row removal) → the operator is non-monotone
- Otherwise (old output rows always remain "correct") → the operator is monotone

R

uid	gid
123	gov
857	abc
189	abc

−

S

uid	gid
123	gov
901	edf

=

uid	gid
857	abc
189	abc

This old row is always valid no matter what rows are added to R

Classification of relational operators

- Selection: $\sigma_p R$ — Monotone
- Projection: $\pi_L R$ — Monotone
- Cross product: $R \times S$ — Monotone
- Join: $R \bowtie_p S$ — Monotone
- Natural join: $R \bowtie S$ — Monotone
- Union: $R \cup S$ — Monotone
- Difference: $R - S$ — Monotone w.r.t. R; non-monotone w.r.t S
- Intersection: $R \cap S$ — Monotone

Why is "—" needed for "highest"?

- Composition of monotone operators produces a monotone query
  - Old output rows remain "correct" when more rows are added to the input
- Is the "highest" query monotone?
  - No!
  - Current highest *pop* is 0.9
  - Add another row with *pop* 0.91
  - Old answer is invalidated

☞So it must use difference!

Why do we need core operator X?

- Difference
  - The only non-monotone operator
- Projection
  - The only operator that removes columns
- Cross product
  - The only operator that adds columns
- Union
  - ?
- Selection
  - ?

Extensions to relational algebra

- Duplicate handling ("bag algebra")
- Grouping and aggregation

☞All these will come up when we talk about SQL

☞But for now we will stick to standard relational algebra without these extensions

Relational Calculus (Optional)

- Relational Algebra: **procedural** language
  - An algebraic formalism in which queries are expressed by applying a sequence of operations to relations.
- Relational Calculus: **declarative** language
  - A logical formalism in which queries are expressed as formulas of first-order logic.
- Codd's Theorem: Relational Algebra and Relational Calculus are essentially equivalent in terms of expressive power.

Relational calculus

User (<u>uid</u> int, *name* string, *age* int, *pop* float)
Group (<u>gid</u> string, *name* string)
Member (<u>uid</u> int, <u>gid</u> string)

- Use *first-order logic (FOL) formulae* to specify properties of the query answer

- Example: Who are the most popular?
  - $\{u.uid \mid u \in User \wedge \neg(\exists u' \in User: u.pop < u'.pop)\}$, or
  - $\{u.uid \mid u \in User \wedge (\forall u' \in User: u.pop \geq u'.pop)\}$

Relational calculus

- Relational algebra = "safe" relational calculus
  - Every query expressible as a safe relational calculus query is also expressible as a relational algebra query
  - And vice versa

- Example of an "unsafe" relational calculus query
  - $\{u.name \mid \neg(u \in User)\}$ → users not in the database
  - Cannot evaluate it just by looking at the database

- A query is *safe* if, for all database instances conforming to the schema, the query result can be computed using only constants appearing in the database instance or in the query itself.

Turing machine

How does relational algebra compare with a Turing machine?

- A conceptual device that can execute any computer algorithm
- Approximates what general-purpose programming languages can do
    - E.g., Python, Java, C++, …

Alan Turing (1912-1954)

http://en.wikipedia.org/wiki/File:Alan_Turing_photo.jpg

Limits of relational algebra

- Relational algebra has no recursion
  - Example: given relation *Friend*(*uid1*, *uid2*), who can Bart reach in his social network with any number of hops?
    - Writing this query in r.a. is impossible!
  - So r.a. is not as powerful as general-purpose languages

- But why not?
  - Optimization becomes undecidable
  - ☞Simplicity is empowering
  - Besides, you can always implement it at the application level, and recursion is added to SQL nevertheless!

Summary

- Part 1: Relational data model
  - Data model
  - Database schema
  - Integrity constraints (keys)
  - Languages (relational algebra, relational calculus, SQL)

- Part 2: Relational algebra – basic language
  - Core operators & derived operators
    (how to write a query)
  - V.s. relational calculus
  - V.s. general programming language

- What's next?
  - SQL – query language used in practice (4 lectures)